VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nástroj pro zobrazování vnitřní reprezentace postavené na systému SUIF

Visualization Tool for Intermediate Representations Based on the System SUIF

2010

Svatoslav Musil

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Orlové 28. dubna 2010 .....

Na tomto místě bych rád poděkoval vedoucímu mé bakalářské práce Ing. Marku Běhálkovi za podnětné připomínky, užitečné rady a čas, který mi věnoval.

Abstrakt

Vnitřní reprezentace programu je mezijazyk vygenerovaný ze zdrojového kódu a obohacený o mnoho dalších užitečných informací. Bohužel je pro člověka tato vygenerovaná vnitřní reprezentace čitelná jen s velkými obtížemi, u složitějších aplikací je to téměř nemožné. Tato bakalářská práce si klade za cíl vytvoření aplikace, která bude umožňovat grafické zobrazení jednoho konkrétního formátu vnitřní reprezentace – formátu SUIF. Práce samotná obsahuje stručný přehled formátu SUIF a popis úpravy funkcionality jeho generátoru. V neposlední řadě je součástí této práce i popis návrhu a implementace zobrazovacího nástroje.

Klíčová slova: SUIF, vnitřní reprezentace, XML, zobrazování

Abstract

Internal representation of the program is an intermediate language generated from source code and enriched with many other useful information. Unfortunately, this internal representation is, for human, legible only with great difficulties, for complex applications it is almost impossible. This thesis aims to create an application that will provide graphical view of an internal representation of one particular format - the SUIF format. The work itself contains brief overview of the SUIF format and description of modification of its generator's functionality. Last but not least, this thesis includes the description of design and implementation of a visualization tool.

Keywords: SUIF, intermediate representation, XML, visualization

Seznam použitých zkratek a symbolů

GUI	–	Graphical User Interface
IR	–	Intermediate Representation
JVM	–	Java Virtual Machine
SMGN	–	SUIF Macro Generator
SUIF	–	Stanford University Intermediate Format
XML	–	eXtensible Markup Language

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této bakalářské práce je návrh a implementace nástroje, který by uživateli umožnil zobrazení již existující vnitřní reprezentace. V této práci se zaměřuji pouze na formát SUIF (Stanford University Intermediate Format) jakožto na jeden ze standardních formátů vnitřní reprezentace. Původní SUIF byl vyvinut v jazyce C/C++. V rámci práce byla použita jeho mutace napsaná v jazyce Java. Tuto práci jsem podle pokynů svého vedoucího vystavěl na zdrojových kódech původně vytvořených jako vzorové řešení pro [3], z níž jsem také čerpal. Všechny zdrojové kódy napsané v jazyce Java, které jsou v rámci této práce použity, byly vytvořeny a otestovány při použití *Java SE Development Kit 6 (Update 20)*.

Překladač je nástroj, který provádí převod ze zdrojového jazyka do jazyka cílového. Překlad samotný pak obvykle sestává ze tří po sobě jdoucích částí (lexikální, syntaktická a sémantická analýza). Pokud překlad během žádné z těchto tří analýz neskončí s chybou, pak je na výstupu překladače výsledná vnitřní reprezentace, nezávislá na zdrojovém či cílovém jazyku. Obvykle se pak ještě optimalizuje a nakonec je z ní vygenerován přímo spustitelný program.

1.1 Přehled kapitol

- *Systém SUIF* - v této kapitole čtenáře seznámím se systémem SUIF. Je zde také obsažen stručný popis jazyka Hoof používaného pro definic tříd vnitřní reprezentace.
- *Doplnění funcionality překladače* - zde popisuji kroky, které bylo nutno provést ještě před zahájením tvorby vlastního vizualizačního nástroje. Jedná se zejména o problematiku předávání informací, nutných pro funkci zobrazovacího nástroje, do uzlů vnitřní reprezentace.
- *Nástroj pro zobrazování vnitřní reprezentace* - tato kapitola obsahuje popis analýzy, návrhu a implementace vizualizačního nástroje. Zde jsem se snažil dodržovat kroky standardního softwarového procesu.

2 Systém SUIF

Na univerzitě Stanford byl navržen a vyvinut systém SUIF, jehož hlavním úkolem bylo zjednodušit a sjednotit proces návrhu a tvorby překladačů. Tento systém poskytuje množinu uzlů vnitřní reprezentace, která postihuje během překladu nejčastěji používané typy objektů (např. třída, metoda apod.) - programátor tedy nemusí vyvíjet vše od začátku, ale má k dispozici základ, který podle svých potřeb může doplnit o další uzly či upravit vlastnosti stávajících uzlů.

2.1 Základní uzly systému SUIF

Vnitřní reprezentace systému SUIF je navržena tak, aby každý uzel rozšiřoval vlastnosti uzlu, který mu je hierarchicky nadřazen. Tyto definice tvoří strom, ve kterém výše postavený uzel definuje obecné vlastnosti, které jeho potomci přejímají a doplňují vlastnosti nové. Doplňováním nových uzlů tedy dochází k postupnému zjemňování a zpřesňování definic.

Hierarchicky nejvyšší třídou v této vnitřní reprezentaci je *SuifObject*. Ten definuje atribut *parent* reprezentující objekt vlastníka uzlu a další metody (např. *deep_clone* nebo *shallow_clone* pro klonování objektů či metody pro tisk objektů).

Třída *AnnotableObject* je základem, na kterém jsou postaveny všechny ostatní uzly vnitřní reprezentace. Z definice vyplývá, že všechny třídy, které z ní dědí, mohou obsahovat anotaci - všechny anotace jsou pak uloženy v proměnné *annotes*.

Každá anotace, kterou bude potřeba připojit do vnitřní reprezentace, musí být potomkem třídy *Annote*. Tyto objekty umožňují programátorovi udržovat ve vnitřní reprezentaci dodatečné informace, které potřebuje. Je rovněž možné připojit anotaci k objektu dědícímu ze třídy *Annote*.

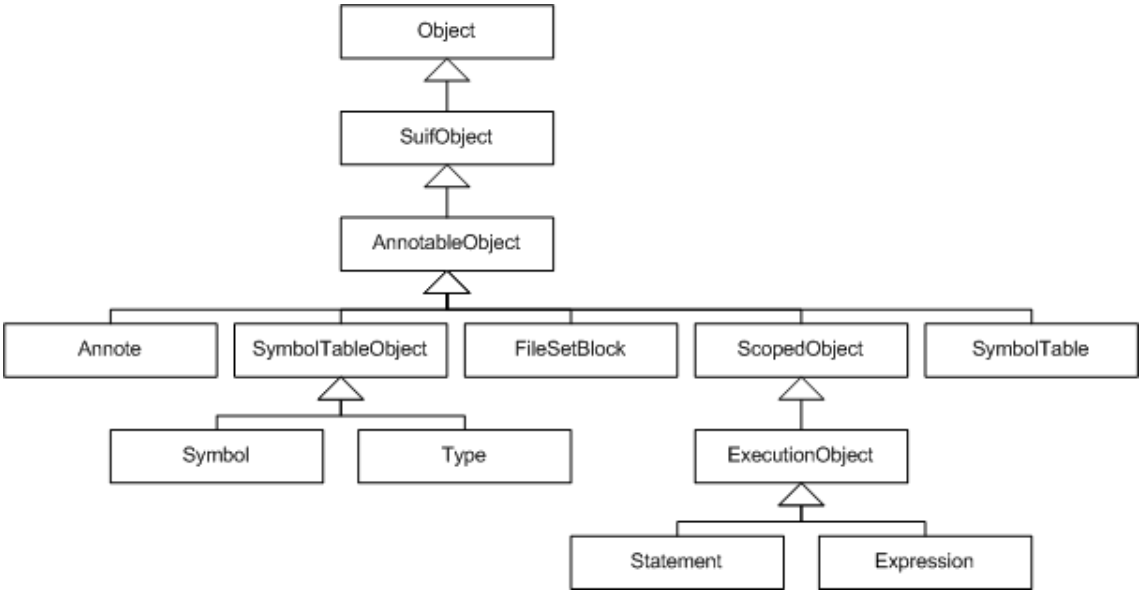
Dalším důležitým typem uzlu je *SymbolTableObject*, který definuje atribut *name* udávající jméno objektu. Jedině potomci této třídy mohou být udržováni v tabulkách symbolů - jedná se o symboly a jejich typy.

Tabulka symbolů je definována jako potomek třídy *SymbolTable*. V systému SUIF neexistuje globální tabulka symbolů, místo toho jsou vytvářeny samostatně v objektech jako jsou např. definice funkcí či tříd.

Všichni potomci třídy *ScopedObject* představují část kódu, která je nějakým způsobem proveditelná - může se jednat např. o výrazy, příkazy či definice funkcí a proměnných.

Kořenovým uzlem vygenerované vnitřní reprezentace je uzel typu *FileSetBlock*, který má v sobě vnořeny objekty typu *FileBlock*, z nichž každý reprezentuje jeden překládaný zdrojový soubor.

Na obrázku 1 je znázorněna struktura několika hierarchicky nejvýše postavených základních uzlů systému SUIF. Kompletní hierarchie všech standardních uzlů vnitřní reprezentace je uvedena v [1, str. 9]. Čtenáře, zajímajícího se o podrobný popis vlastností standardních uzlů vnitřní reprezentace, by mohla zajímat příručka [2], která toto popisuje naprosto vyčerpávajícím způsobem.



Obrázek 1: Základní uzly systému SUIF

2.2 Definice uzlů v jazyce Hoof

Definice tříd vnitřní reprezentace je tvořena několika soubory napsanými v jazyce Hoof. Tento přístup může programátorovi ušetřit poměrně hodně času, protože namísto přímé tvorby tříd si pouze nadefinuje, jaké třídy bude používat, uvede názvy proměnných a jejich datové typy a poté spustí překlad, který je proveden aplikací *SUIF Macro Generator*.

Každý .hoof soubor může obsahovat jeden nebo více modulů, přičemž třídy definované v jednom modulu jsou vygenerovány do stejného balíčku. Definice modulu se dá obecně zapsat takto:

```
module název_modulu {
    modifikátor  název_nové_třídy : název_nadřazené_třídy {
        typ_prvku * typ_odkazu název_proměnné;
        typ_kolekce<typ_prvku * typ_odkazu> název_kolekce;
    }
}
```

Výpis 1: Syntaxe souboru v jazyce Hoof

Zde jsou popsány názvy použité při popisu syntaxe:

- *název_modulu* - udává název balíčku, do kterého budou patřit výsledné vygenerované třídy
- *modifikátor* - klíčové slovo *abstract* nebo *concrete*
- *název_nové_třídy* - pro třídu bude vygenerován soubor s tímto názvem a příponou .java

- *název_nadřazené_třídy* - třída, kterou tato rozšiřuje
- *typ_prvku* - základní datový typ (např. *String*, *float* nebo *int*)
- *typ_odkazu* - klíčové slovo *owner* nebo *reference*, které určuje, jestli je daný uzel v této třídě definován nebo zda se na něj třída pouze odkazuje
- *název_proměnné* - při překladu je název proměnné upraven tak, že všechna velká písmena jsou převedena na malá a je před ně vložen znak podtržítka
- *typ_odkazu* - definice objektu, který udržuje seznam odkazů nebo definic. Popis použitých tříd je uveden v tabulce 1
- *název_kolekce* - platí pro něj stejná pravidla, jako pro *název_proměnné*

2.3 SUIF Macro Generator

SUIF Macro Generator (zkráceně SMGN) je aplikace, která ze zdrojových souborů napsaných v jazyce Hoof vygeneruje množinu zdrojových souborů v jazyce Java. Při překladu zároveň vznikne soubor s příponou .boof, který obsahuje binární reprezentaci konkrétního modulu. Pokud je tento modul využíván jiným modulem, je potřeba tento binární soubor zkopírovat do složky se zdrojovým kódem importujícího modulu.

Mějme následující příklad definičního souboru jazyka Hoof:

```
module example {
    concrete ExampleObject : AnnotableObject {
        String name;
        searchable_list<AnnotableObject> objects;
    };
}
```

Výpis 2: Příklad definičního souboru v jazyce Hoof

Úvodní část *module example* říká, že všechny třídy, které jsou definovány uvnitř složených závorek, budou patřit do jednoho balíčku - v našem případě bude mít každý vygenerovaný soubor na prvním řádku uvedeno *package suif_j.example;*. Principiálně je možné definovat několik modulů v jednom zdrojovém souboru, obvykle se však tato možnost z důvodu přehlednosti nepoužívá a každý modul je uveden v odděleném souboru. Jelikož SMGN používá návrhový vzor továrna, je mezi vygenerovanými soubory v tomto případě i *ExampleObjectFactory.java*.

V příkladu definuji jedinou třídu *ExampleObject*, která rozšiřuje *AnnotableObject* a má atribut *name* typu *String*. Ke každé proměnné základního datového typu, která je v rámci jedné třídy definována, jsou vygenerovány odpovídající *get* a *set* metody. Každá třída může být definována s modifikátorem *abstract* nebo *concrete*.

Pokud programátor potřebuje použít kolekce, může k tomu použít klíčová slova uvedená v tabulce 1. Podle použitého klíčového slova je vytvořena proměnná a vygenerovány potřebné přístupové metody (konkrétně pro *searchable_list* uvedený v příkladu by jednou z nich byla např. metoda *void append_object(AnnotableObject elem)*, zajišťující vložení

Klíčové slovo	Použitá třída
list	*java.util.LinkedList*
searchable_list	*java.util.ArrayList*
indexed_list	*java.util.HashMap*
vector	*java.util.Vector*

Tabulka 1: Klíčová slova jazyka Hoof pro kolekce v jazyku Java

prvku *elem* do proměnné *objects*. Podrobný popis generovaných metod a celkově i jazyka Hoof je uveden v [6].

Pro úplnost uvádím ještě soubor *ExampleObject.java*, který by vznikl překladem výše uvedeného příkladu definičního souboru jazyka Hoof:

```
package suif_j.example;
import java. util .*;
import java.io .*;

public class ExampleObject extends AnnotableObject implements Serializable {
    public String  get_name() {
        return this._name;
    }
    public void set_name(String the_value) {
        this._name = the_value;
    }
    protected String _name;
    public void append_object(AnnotableObject elem) {
        _objects.add(elem);
    }
    public  Iterator   get_object_iterator ()  {
        return _objects. iterator () ;
    }
    public void remove_object(AnnotableObject elem) {
        _objects.remove(elem);
    }
    public boolean has_object_member(AnnotableObject elem) {
        return _objects.contains(elem);
    }
    public int  get_object_count() {
        return _objects.size () ;
    }
    public void insert_object( int  pos, AnnotableObject elem) {
        _objects.add(pos, elem);
    }
    public AnnotableObject remove_object(int pos) {
        return (AnnotableObject)_objects.remove(pos);
    }
    public AnnotableObject get_object(int pos) {
        return (AnnotableObject)_objects.get(pos);
    }
    protected List _objects = new ArrayList();
```

```
    public void print(FormattedText out) {
        out.start_block("ExampleObject",this);
        out. print_list ("AnnotableObject.annotes",_annotes,true);
        out. print_field ("ExampleObject.name",_name,false);
        out. print_list ("ExampleObject.objects",_objects,false);
        out.end_block("ExampleObject");
    }
}
```

Výpis 3: Příklad vygenerovaného souboru *ExampleObject.java*

2.4 Jádro systému SUIF

Jednou z hlavních tříd celého jádra je *SuifEnv*, které představuje samotné prostředí překladače. Toto prostředí udržuje aktuální stav a komponenty celého systému, žádná další globální proměnná zde neexistuje.

Další důležitou třídou definovanou v jádře je *SuifObject*, jehož popis i popis uzlů na něm postavených byl již uveden v kapitole 2.1.

Jelikož je celé jádro systému SUIF implementováno v jazyce Java, jsou pro vstupně/výstupní operace použity jeho standardní metody. To v praxi znamená, že všechny uzly vnitřní reprezentace implementují rozhraní Serializable, čímž je umožněno celou vnitřní reprezentaci serializovat a uložit do binárního souboru s příponou .suif.

Další možností uložení vnitřní reprezentace je zapsat ji do souboru ve formátu XML. Metody pro výstup v tomto formátu jsou definovány v balíku *kernel*, konkrétně v souboru *FormattedText.java*.

3 Doplnění funkcionality překladače

Uživatel potřebuje mít možnost specifikovat uzly, které bude chtít zobrazovat. Jelikož si může nadefinovat vlastní uzly či dokonce upravovat uzly standardní, bylo potřeba navrhnout nějaký systém, kterým by se do vnitřní reprezentace uložily informace související s jejím zobrazováním. K tomuto účelu jsem v modulu *basic* vytvořil novou třídu *ViewerAnnote*, jejíž zdrojový kód je uveden ve výpisu 4.

```
concrete ViewerAnnote : Annote {
    String name implements name;
    String extendsType;
    int type;
    String id;
    boolean displayable;
    searchable_list<String> defaultNodes;
};
```

Výpis 4: Definice třídy *ViewerAnnote*

ViewerAnnote rozšiřuje standardní třídu *Annote* a má následující atributy:

- *String name* - protože třída *Annote* je definována jako abstraktní a obsahuje tento atribut, je nutné jej implementovat. Já v něm uchovávám název třídy, ke které se tato anotace vztahuje.
- *String extendsType* - název třídy, která je přímo hierarchicky nadřazená této třídě. Toto je zavedeno z toho důvodu, abych byl schopen postihnout dědičnost každého typu uzlu.
- *int type* - proměnná obsahující informaci o tom, jak se má daný uzel zobrazit. Platné hodnoty jsem definoval ve třídě *SuifNodeConst*, rovněž umístěné v balíku *kernel*.
- *String id* - název proměnné, ve které je uloženo jméno objektu. Pokud je třeba načíst je z jiného uzlu, lze toho dosáhnout pomocí tečkové konvence, kdy část za poslední tečkou je název této proměnné a všechny předchozí části oddělené tečkou jsou uzly, které je nutno projít zleva doprava, aby bylo možno toto jméno načíst (např. *type.name* říká, že v objektu definovaném či referovaném pod názvem *type* je název objektu uložen v proměnné *name*).
- *boolean displayable* - určuje, jestli se má uzel tohoto typu nabízet ve výběru zobrazovaných uzlů. Pokud je nastaveno na *true*, jsou automaticky zobrazovány i všechny hierarchicky podřízené uzly.
- *searchable_list<String> defaultNodes* - seznam názvů uzlů, které se mají zároveň s tímto uzlem zobrazit také. Opět je možno použít tečkovou notaci jako v případě *id*, kde část za poslední tečkou je zobrazovaný uzel. Všechny uzly, přes které tento odkaz prochází, jsou zobrazeny také.

Uživatel strukturu této třídy znát nemusí, protože s ní nepracuje přímo, ale pouze pomocí anotace *@SuifNode*, kterou jsem přidal do balíku *kernel* a jejíž definice je uvedena ve výpisu 5. Tyto anotace je pak možno přidat na řádek předcházející definici třídy v definičním souboru jazyka Hoof. Při zpracování tohoto souboru je pak anotace přenesena do vygenerovaného zdrojového souboru v jazyce Java a během překladu přiložena k výsledné třídě.

S touto anotací pak dále pracuje překladač systému SUIF a podle informací, které jsou v ní uloženy, přidává do vnitřní reprezentace odpovídající vygenerované anotace typu *ViewerAnnote*. Proces generování těchto anotací je popsán dále v části 3.2.

```
package suif_j.kernel;
import java.io.*;
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SuifNode
{
    int type() default SuifNodeConst.TYPE_UNDEFINED;
    String id() default "";
    String [] defaultNodes() default {};
    boolean display() default false;
}
```

Výpis 5: Definice anotace *SuifNode*

Jelikož je potřeba, aby byly všechny uživatelsky definované anotace dostupné zejména za běhu programu, použil jsem typ anotace *@Retention(RetentionPolicy.RUNTIME)*, který toto zajišťuje. Atributy *id* a *type* odpovídají stejně pojmenovaným atributům, které jsem definoval pro třídu *ViewerAnnote*, atribut *display* odpovídá atributu *displayable*. Uživatel může zadat seznam uzlů, které chce mít pro daný typ zobrazeny jako výchozí, jako pole řetězců s názvem *defaultNodes*.

Všechny atributy, které tato anotace definuje, mají své výchozí hodnoty - ty jsem zavedl z toho důvodu, aby uživatel nebyl nucen nastavovat všechny hodnoty, ale pouze ty, které jsou pro něj v danou chvíli důležité.

Používané číselné konstanty určující typ uzlu jsou opět definovány v balíku *kernel*, a to v souboru *SuifNodeConst.java*. Tyto hodnoty silně doporučuji neměnit, jelikož jsou nakopírovány přímo do zobrazovacího nástroje a jejich modifikace způsobí nekonzistenci a následně nekorektní chování. Pokud je z nějakého důvodu potřeba je upravit, je možné tak učinit, ale v tom případě je bezpodmínečně nutné tyto změny přenést i do zdrojového kódu zobrazovacího nástroje a ten pak znovu přeložit. Jak jsem však předeslal, uživatel se pak vystavuje nebezpečí, že se tento zobrazovací nástroj přestane chovat korektně.

3.1 Podpora anotací překladače jazyka Hoof

Protože jazyk Hoof oproti Javě nepodporuje anotace, bylo nutno tuto podporu doplnit. K tomuto účelu jsem do třídy *Construct* z balíku *smgn* přidal zdrojový kód uvedený ve výpisu 6. Metoda *addAnnote* pouze uloží svůj parametr (řetězec *annote*) do kolekce *annotes*.

```
Vector annotes = new Vector();

void addAnnote(String annote)
{
    annotes.add(annote);
}
```

Výpis 6: Zdrojový kód přidaný do třídy *Construct*

Do metody *void output(String output_dir)* ve třídě *Construct* jsem přidal řádky uvedené ve výpisu 7. Tato část zdrojového kódu zajistí, že pokud byly k dané třídě připojeny nějaké anotace, budou na korektním místě (tj. na řádcích před začátkem definice třídy) vytištěny.

```
Enumeration<String> ann = annotes.elements();

while (ann.hasMoreElements())
{
    String s = ann.nextElement();

    out.print(s);
}
```

Výpis 7: Zdrojový kód přidaný do metody *output* třídy *Construct*

V této fázi jsem měl připraveny metody pro předání anotací z definičních souborů v jazyce Hoof do vygenerovaných tříd v jazyce Java. Z tohoto důvodu jsem do překladače jazyka Hoof (soubor *HoofParser.jj*) musel zavést rozpoznávání terminálů *ANNOTE* ve formátu anotací jazyku Java, které začínají znakem @ a končí znakem konce řádku[1]. V neterminálu *construct_declaration* pak překladač v cyklu prochází všechny nalezené anotace a přidává je do třídy použitím výše uvedené metody *addAnnote*. Všechny předané anotace jsou poté při generování výsledných tříd představujících uzly vnitřní reprezentace zapsány do výsledného souboru.

3.2 Generování uzlů IR podle zadaných anotací

Dále bylo nutné přimět překladač systému SUIF, aby mnou definované anotace (resp. objekty z těchto anotací vytvářené) vytvářel a připojoval na korektní místa, konkrétně do proměnné *annotes*. K tomuto účelu jsem definoval metodu *void addViewerAnnote()* a protože je potřeba, aby byla dostupná z každého uzlu vnitřní reprezentace, vložil jsem ji přímo do definice třídy *AnnotableObject* v modulu *basic*. Jelikož slovní popis procesu

[1]Specifikace jazyka Java sice umožňuje anotace psát na více řádků, pro účely této práce však postačí anotace jednořádkové.

připojování anotací nemusí být úplně zřejmý, je na obrázku 2 zakreslen odpovídající aktivitní diagram.

Tato metoda nejprve ověří, zda se nejedná o typ uzlu *ViewerAnnote* a pokud ano, pak skončí, protože není žádný důvod k němu připojovat další anotace tohoto typu. Poté je ověřeno, jestli už tento uzel nebyl jednou zpracován (tzn. že ještě neobsahuje žádné anotace typu *ViewerAnnote*) a pokud ano, skončí. Poté si uloží odkaz na objekt reprezentující aktuální třídu, ze které se pokusí získat připojenou anotaci typu *SuifNode*. Pokud ji obsahuje, zjistí si její atributy a vytvoří novou instanci objektu typu *ViewerAnnote*. Pokud výše zmíněná anotace připojena nebyla, je vytvořena „prázdná“ instance objektu typu *ViewerAnnote*. Instance objektu reprezentujícího anotaci je poté připojena k aktuálnímu uzlu.

Tento scénář se opakuje postupně pro všechny hierarchicky nadřazené třídy aktuálního uzlu a skončí až u třídy typu *SuifObject*, protože ten už žádné anotace typu *SuifNode* obsahovat nemůže[2]. Všechny objekty typu *ViewerAnnote*, které jsou k aktuálnímu uzlu připojeny, jsou tedy vytvářeny v pořadí od aktuálního uzlu až k *AnnotableObject*, přičemž jako první je vložen popis aktuálního uzlu.

3.3 Příklad použití uživatelských anotací

Ve výpise 8 je uveden[3] příklad možného použití anotací *@SuifNode*. Kompletní zdrojové kódy níže uvedeného příkladu jsou dostupné na přiloženém CD v modulu *basic*.

```
@SuifNode(id="name")
abstract SymbolTableObject : AnnotableObject {
  String name default <* "" *>;
};

@SuifNode(type=SuifNodeConst.TYPE_DATATYPE)
abstract Type : SymbolTableObject
{ };

@SuifNode(display=true, defaultNodes={"type"})
abstract Symbol : SymbolTableObject
{
  boolean is_address_taken default <* false *>;
  virtual  Type * reference type;
};

@SuifNode(defaultNodes={"definition"}, type=SuifNodeConst.TYPE_VARIABLE)
concrete VariableSymbol : Symbol
{
  QualifiedType* owner type implements type;
   VariableDefinition  * reference definition  omitted;
};
```

[2]To není úplně přesné, uživatel by ji mohl manuálně přidat do zdrojového souboru *SuifObject.java* z balíku *kernel*, ale ani v tom případě by se změna do výsledné IR nepromítla, protože třída *SuifObject* není potomkem třídy *AnnotableObject*.

[3]Z tohoto výpisu jsem z důvodu přehlednosti odstranil kód, který vykonává nějakou činnost a ponechal pouze definice členských proměnných uvedených tříd, na jejichž popis se budu v tuto chvíli soustředit.

```
@SuifNode(type=SuifNodeConst.TYPE_PROCEDURE)
concrete ProcedureSymbol : Symbol
{
  ProcedureType* owner type implements type;
  ProcedureDefinition * reference definition  optional;
};

@SuifNode(display=true, defaultNodes={"variable_symbol"}, id="variable_symbol.name")
concrete VariableDefinition  :  ScopedObject
{
  VariableSymbol* reference variable_symbol;
  int  bit_alignment;
  ValueBlock* owner  initialization ;
  boolean is_static default <* false  *>;
};

@SuifNode(id="procedure_symbol.name")
concrete ProcedureDefinition : ScopedObject
{
  ProcedureSymbol* reference procedure_symbol; # build <* null *>;
  ExecutionObject* owner body; # build <* null *>;
  SymbolTable* owner symbol_table
  build <* BasicObjectFactory.create_basic_symbol_table(null) *>;
  DefinitionBlock* owner definition_block
  build <* BasicObjectFactory.create_definition_block() *>;
   list <ParameterSymbol* reference> formal_parameters;
};
```

Výpis 8: Příklad použití uživatelských anotací

Příklad nejprve definuje třídu *SymbolTableObject*. Má jediný atribut *name*, který připojená anotace určuje jako proměnnou, ve které je uloženo jméno daného objektu.

Druhou uvedenou třídou je *Type*, která nedefinuje žádné nové vlastnosti. Pomocí anotace je tato třída označena jako datový typ.

Následuje definice třídy *Symbol*, která obsahuje poduzel *type* typu *Type*. Klíčové slovo *reference* říká, že tento poduzel je definován v jiném bloku. Připojená anotace určuje, že všechny objekty tohoto typu se mají zobrazit ve výchozím nastavení, přičemž bude automaticky zobrazen i poduzel *type*.

Třída *VariableSymbol* dědí všechny údaje specifikované anotací třídy *Symbol*, na které je založena. Navíc anotace k ní připojená říká, že se ve výchozím nastevní zobrazí i poduzel *definition*. Tato třída i všechny třídy z ní dědící budou považovány za proměnnou.

Dále uvedená třída *ProcedureSymbol* přejímá všechny vlastnosti třídy *Symbol*, včetně výchozího zobrazení poduzlu *type*, u nějž klíčové slovo *owner* navíc říká, že je zde přímo definován. Navíc je tato třída považována za proceduru.

Objekty typu *VariableDefinition* budou přímo nabízeny k zobrazení. Aby bylo při zobrazení zřejmé, ke které proměnné se tato definice váže, je jako jméno tohoto objektu použito jméno definované proměnné. Ve výchozím nastavení bude zobrazen také referovaný poduzel *variable_symbol*.

Poslední zde uvedená třída *ProcedureDefinition* načítá své jméno podobně jako třída *VariableDefinition* z objektu, ke kterému se vztahuje.

Obrázek 2: Aktivitní diagram procesu připojování anotací

4 Nástroj pro zobrazování vnitřní reprezentace

V této části se budu zabývat návrhem a implementací samotného nástroje na zobrazování vnitřní reprezentace. Rozhodl jsem se napsat jej celý v jazyce Java, protože je v ní napsán celý systém SUIF a také proto, že mám s tímto programovacím jazykem asi největší zkušenosti.

Jak již bylo řečeno, vnitřní reprezentace je uložena ve dvou samostatných souborech, a to binárním s příponou .suif a textovým s příponou .xml. Jelikož Java nabízí standardní prostředky pro práci s formátem XML, bude zobrazovací nástroj pracovat pouze s ním. Důvodem je relativně snadná kontrola správnosti zobrazovaných informací, kdy může uživatel obeznámený se systémem SUIF jednoduše porovnat informace zobrazené a informace uložené v souboru. Toto by nebylo možné při použití binárního formátu.

4.1 Specifikace zadání

Protože každý uživatel má jiné požadavky na vlastnosti používaného softwarového produktu, není možné vyhovět všem. Základní očekávaná funkcionalita by se dala shrnout do několika bodů:

- Možnost mít otevřeno více dokumentů najednou.
- Program musí ke své funkci vyžadovat minimální uživatelský vstup. Pokud už je potřeba, aby uživatel zadal nějaké doplňující informace, mělo by být možné to udělat interaktivně (tím je myšleno, že uživatel nemusí nic vypisovat - vše musí být možno „naklikat“).
- Uživatel musí mít v otevřeném dokumentu možnost vybírat uzly, které chce zobrazit. To by mělo být možné nejen podle typu uzlu, ale i podle jeho umístění ve stromové struktuře.
- Je nutné rozlišovat mezi uzly, které jsou v daném místě definované a které jsou pouze referované. Tento rozdíl musí být patrný na první pohled, aby nedocházelo k matení uživatele.
- Jelikož je každý uzel potomkem několika tříd, musí být výsledný program schopen upravit zobrazení uzlu vnitřní reprezentace podle toho, jak na něj uživatel v dané chvíli potřebuje nahlížet (např. na uzel typu *ClassType* lze nahlížet také jako na uzel typu *DataType*). To v praxi znamená, že je nutné zachytávat informace o dědičnosti každého typu uzlu.
- Uživatel potřebuje mít možnost porovnat dvě vnitřní reprezentace, a to jak z hlediska celého stromu definic, tak pro dva konkrétní uzly. Program musí výsledky porovnání vypsat stručně a přitom srozumitelně.

4.2 Prostředí pro implementaci

Zobrazovací nástroj musí být schopen graficky a pokud možno i přehledně zobrazit strukturu uloženou v souboru, který zadá uživatel. Z tohoto důvodu je nemožné, aby bylo zobrazování prováděno v konzolové aplikaci - a protože je k implementaci použita Java, opět mám možnost využití jejich standardních prostředků pro práci s grafikou, konkrétně bude použita sada komponent z balíku *Swing*.

Dále bylo nutno si zvolit způsob, jakým budou zpracovány XML soubory. Java nabízí dvě základní možnosti, jak tyto dokumenty zpracovat:

- *SAX (Simple API for XML)* - jedná se o balík tříd, které zpracovávají XML dokument sekvenčně. Jsou poměrně rychlé a mají nízké paměťové nároky, ale při jejich použití je nemožné se při čtení dokumentu vracet.
- *DOM (Document Object Model)* - třídy z tohoto balíku přistupují k XML souboru tak, že si jej celý načtou do paměti ve formě objektů a s těmito objekty pak uživatel pracuje. Jsou o něco pomalejší a mají vyšší paměťové nároky než SAX, ale na druhou stranu umožňují náhodný přístup ke všem prvkům v paměti.

Pro implementaci svého programu jsem si vybral balík *DOM*, protože práce s ním se mi zdá mnohem jednodušší než v případě *SAX*. Jedinou nevýhodou použitého řešení je potenciální možnost vyčerpání přidělené paměti či přetečení zásobníku u rozsáhlejších projektů. S ničím takovým jsem se však během implementace nesetkal.

Pokud by tedy došlo k vyvolání výjimky *java.lang.OutOfMemoryError*, je možné zvětšit množství paměti, které JVM (*Java Virtual Machine*) přiděluje. Druhá potenciálně možná výjimka *java.lang.StackOverflowError* by mohla být vyvolána u dokumentů, kde je příliš velké vnoření uzlů - řešením může být nastavení většího množství paměti používané zásobníkem.

4.3 Specifikace požadavků

Podle zadání uvedeného v části 4.1 a částečně i z dříve popsaného postupu přidávání uživatelských anotací (viz. kapitola 3) jsem nyní schopen říci, že s výsledným vizualizačním nástrojem budou pracovat tři základní typy uživatelů:

- *Uživatel* - jedná se o člověka, který pracuje pouze s vygenerovanou vnitřní reprezentací. Teoreticky není nutné, aby měl nějaké speiální znalosti z oblastí jako tvorba překladačů nebo programování.
- *Programátor* - jedná se o uživatele, který má navíc znalosti týkající se programování. Jeho úloha obvykle spočívá v napsání zdrojového kódu programu a jeho následném přeložení do vnitřní reprezentace, kterou mu poskytnul tvůrce překladače. Vizualizační nástroj bude používat k ověření, že se jím napsaný program přeloží do očekávané struktury.

- *Tvůrce překladače* - může se jednat o programátora, který zná systém SUIF, je obeznámen s jazykem Hoof a dokáže podle svých potřeb modifikovat stávající nebo vytvářet další potřebné třídy vnitřní reprezentace. Nástroj pro zobrazování IR používá k ověření správnosti jím definovaných tříd.

Výsledný vizualizační nástroj bude natolik úzce spjatý s činností výše uvedených uživatelů, že není důvod jejich funkci nějak striktně oddělovat. Diagram 3 ukazuje nejen jejich interakci se zobrazovacím nástrojem, ale také se samotným systémem SUIF. Pro přehlednost jsem tento diagram rozdělil do tří samostatných oblastí odpovídajících tomu, jak postupuje práce těchto uživatelů od návrhu tříd vnitřní reprezentace, přes jejich použití až po konečné zobrazení výsledné IR.

4.4 Analýza požadavků

Tato část obsahuje podrobnější rozbor požadavků, které jsem shrnul v předchozích třech částech. Některé pasáže, pokud jsou dostatečně zřejmé, jsem zde nepopisoval. V jiném úseku se naopak mohou dvě specifikace prolínat, pakliže jsou úzce spjaty.

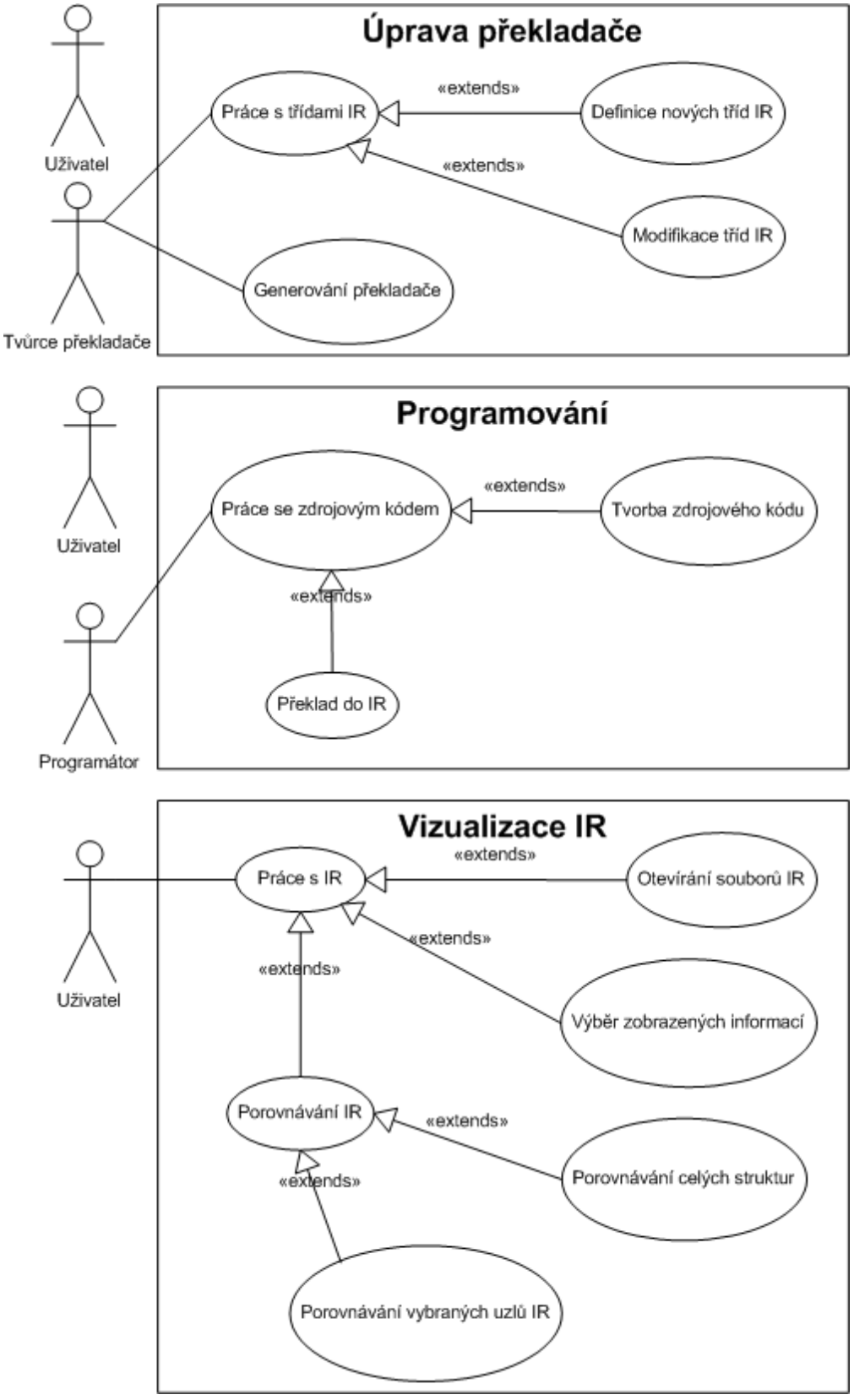
4.4.1 Otevřené dokumenty

Prvním požadavkem byla možnost mít otevřeno více dokumentů najednou. Jelikož se budu na tyto otevřené soubory za běhu programu odkazovat, musím tento seznam nějak udržovat. Určitě nemá příliš smysl ukládat jej jinam než do hlavního formuláře, který je v paměti po celou dobu běhu programu. Přesněji řečeno, pro seznam otevřených dokumentů bude vytvořena jednoduchá třída, která bude toto obsluhovat. Instance objektu této třídy bude pak uložena přímo v hlavním formuláři programu, což je dobře vidět na obrázku 4, přičemž seznam dokumentů bude udržován podle jména souboru. Z toho pak vyplývá, že každý dokument může být otevřen nejvýše jednou.

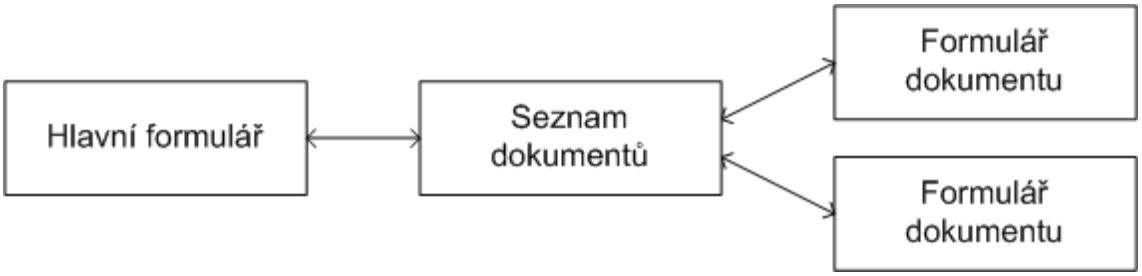
4.4.2 Struktura načtených informací

Principiálně bude nutné udržovat v paměti nejméně dva typy informací o načtené struktuře dokumentu:

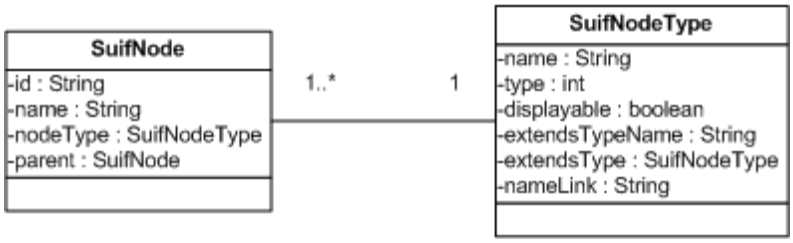
- **Typ uzlu** - jedná se o informace, které jsou společné pro všechny uzly stejného typu jako např. jeho název a uživatelsky definovaný typ či odkaz na atribut obsahující jméno uzlu. Zde je také udržován seznam všech uzlů daného typu.
- **Konkrétní uzel** - zde jsou uloženy informace, které se vztahují ke konkrétní instanci uzlu vnitřní reprezentace. Obsahuje zejména načtený název a identifikátor uzlu, odkaz na uzel, ve kterém je tento definován a odkaz na objekt reprezentující jeho typ. Názvy uzlů jsou načítány podle odkazů, které specifikuje uživatel pomocí anotací. Pokud toto nedefinuje, pak je jméno objektu tvořeno názvem jeho typu a číslem určujícím pořadí, ve kterém byl tento uzel vnitřní reprezentace načten.



Obrázek 3: Možné případy užití



Obrázek 4: Schéma udržování seznamu dokumentů



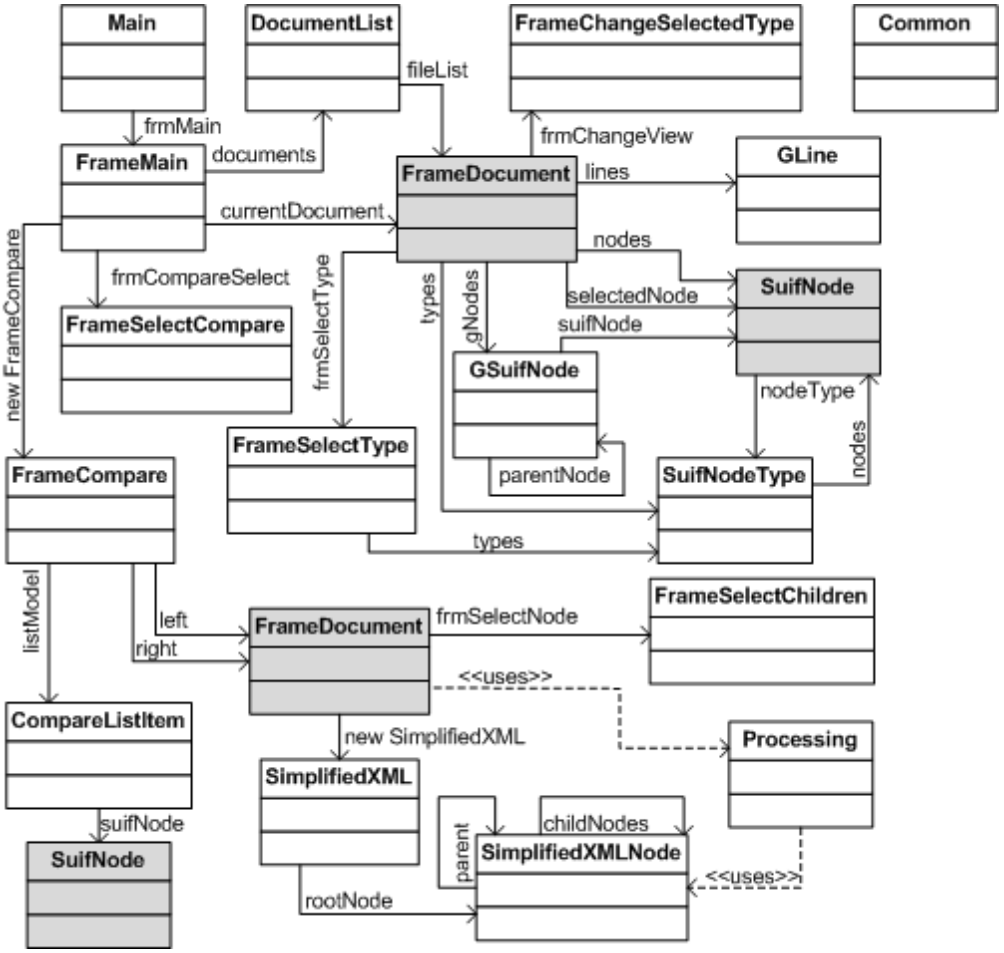
Obrázek 5: Vztah mezi uzlem vnitřní reprezentace a jeho typem

4.4.3 Struktura aplikace

Členění aplikace z hlediska tříd bylo částěčně nastíněno již v předchozí části, kompletní struktura[4] je uvedena na třídním diagramu 6. Následuje stručný popis některých tříd:

- *FrameMain* - hlavní formulář celého programu. Obsahuje seznam otevřených dokumentů, zvlášť uchovává odkaz na aktuálně zobrazený formulář dokumentu. Dále obsahuje systém nabídek pro ovládání programu.
- *DocumentList* - seznam aktuálně otevřených souborů.
- *FrameDocument* - každý objekt tohoto typu obsahuje právě jeden otevřený a zobrazený dokument.
- *Processing* - tato třída se stará o transformaci zjednodušené struktury XML souboru do struktury, kterou je výsledný program schopen zobrazit.
- *SuifNode* - jeden objekt tohoto typu představuje jeden uzel vnitřní reprezentace.
- *SuifNodeType* - udržuje informace společné pro uzly stejného typu.
- *GSuifNode* - grafická podoba konkrétního uzlu vnitřní reprezentace. Těchto grafických podob může být více, podle četnosti použití daného uzlu.

[4] Z hlediska velkého počtu proměnných a metod v celé aplikaci jsem se rozhodl je v tomto diagramu neuvádět - vedlo by to jen k znepřehlednění už tak poměrně složitého diagramu.



Obrázek 6: Třídní diagram aplikace

- *GLine* - pouze uchovává souřadnice spojnice, která bude vykreslena mezi dvěma zobrazenými uzly vnitřní reprezentace.
- *SimplifiedXML* - zjednodušená podoba XML souboru.
- *SimplifiedXMLNode* - uzel zjednodušené struktury XML souboru.

V tomto třídním diagramu bylo nutné z důvodu nedostatku místa některé třídy zakreslit dvakrát - tyto třídy jsou pak barevně odlišeny. Třída *Common* je zakreslena poněkud stranou, protože je používána několika jinými třídami a zanesení všech spojnic znázorňujících její použití by znamenalo značně zkomplikovat tvorbu i čtení tohoto diagramu.

4.5 Návrh výsledného programu

Jak jsem předeslal v úvodu, pokusím se během celé této práce dodržet standardní softwarový proces. Jelikož základní analýza byla dokončena, můžu se nyní pustit do návrhu výsledné aplikace.

4.5.1 Načítání dokumentů

Bylo nutno si důkladně rozmyslet, jakým způsobem budu načítat soubory vnitřní reprezentace. Již dříve jsem určil, že použiji standardní prostředky jazyka Java, konkrétně knihovnu DOM pro načítání souborů ve formátu XML.

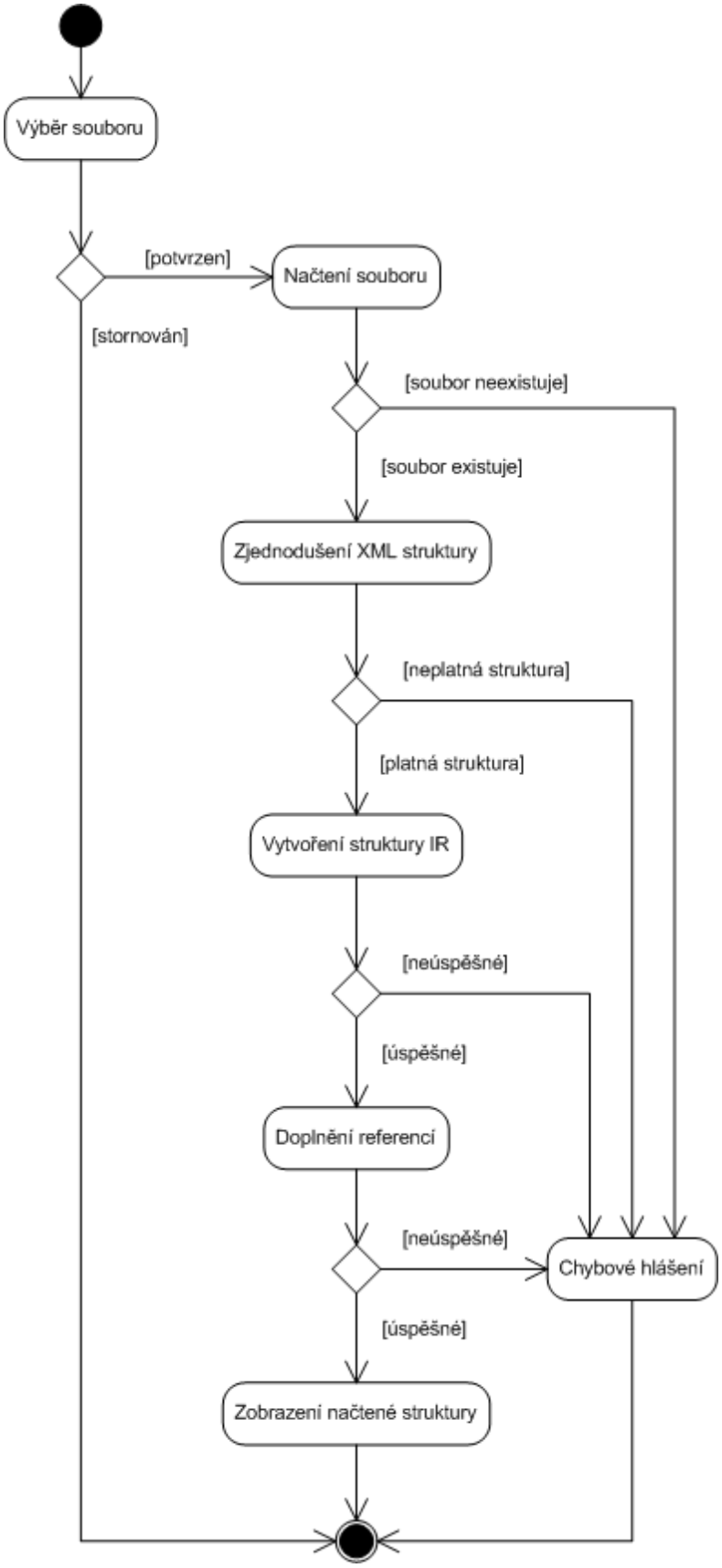
Samotné načítání není triviální záležitost jednak proto, že se během načítání budu muset různě vracet a analyzovat právě načtené informace a jednak proto, že v celé vnitřní reprezentaci se může nacházet mnoho různých referencí. Tyto reference pak nejsem schopen postihnout během jediného průchodu, proto jsem proces vlastního načítání navrhl takto:

- **Načtení XML dokumentu.** Přímé načítání vnitřní reprezentace ze souboru ve formátu XML není právě jednoduchý proces, proto si nejprve přečtu celý dokument, vypreparuji z něj informace, které budu potřebovat a uložím si je do paměti pro další použití. V této fázi budu ignorovat XML komentáře, protože z hlediska načítané vnitřní reprezentace nejsou relevantní.
- **Zpracování vnitřní reprezentace.** Pokud proběhlo načtení XML dokumentu z předchozího kroku bez chyb, mám nyní v paměti jeho zjednodušenou strukturu. Tuto strukturu poté projdu a vytvořím z ní objekty, které budou představovat celou vnitřní reprezentaci. Tyto objekty budou uloženy přímo ve formuláři, který představuje daný dokument.
- **Doplnění již načtených dat.** Jelikož při vytváření objektů ještě nejsem schopen zpracovat všechny reference, které jsou v konkrétní vnitřní reprezentaci uloženy, musím nyní tyto reference doopravit. Běžně se může jednat např. o situaci, kdy narazím na referenci objektu, který je definován v prozatím nezpracovaném bloku. Po úspěšném dokončení této fáze mám tedy načtenou celou vnitřní reprezentaci, která již může být zobrazena.

Pro lepší představu o tom, jak bude celé načítání struktury dokumentu probíhat, jsem vytvořil diagram aktivit, ze kterého by měly být všechny výše uvedené kroky patrné (viz. obrázek 7). Podrobnější popis načítání dokumentů je uveden dále v části 4.6.4.

4.5.2 Zobrazení uzlů vnitřní reprezentace

Každý uzel vnitřní reprezentace bude vykreslen jako obdélník, do něhož budou vepsány základní informace jako např. typ uzlu nebo jeho název. Uzly, které jsou s tímto uzlem v přímém vztahu (tj. jsou v něm definovány nebo jsou tímto uzlem referovány) s ním budou spojeny čarou, aby bylo na první pohled zřejmé, že je mezi nimi nějaký vztah.



Obrázek 7: Aktivitní diagram průběhu načítání vnitřní reprezentace

Obrázek 8: Příklad různého zobrazení téhož uzlu IR

Z důvodu snadného rozlišování mezi uzlem v daném místě definovaným a referovaným jsem se rozhodl tyto rozdíly postihnout graficky. Tím je myšleno, že definovaný uzel bude vykreslen jako obyčejný obdélník, zatímco uzel referovaný bude mít navíc zaoblené rohy. Tento rozdíl je dobře vidět na obrázku 8, kde vlevo je uzel definovaný, zatímco vpravo referovaný.

Aby bylo možno zjišťovat informace o uzlu, který uživatel vybere klepnutím myší, musí si každý grafický uzel držet odkaz na „skutečný“ uzel vnitřní reprezentace. Aktuálně vybraný uzel bude také nutno graficky odlišit od ostatních.

4.5.3 Porovnávání struktur

Jelikož jednou z funkcí programu je i porovnávání vnitřních reprezentací či jejich částí. Pro tuto součást programu jsem vytvořil univerzální formulář s názvem *FrameCompare*, který je založen na třídě *JDialog*. Tím je také při použití modálnosti dosaženo toho, že uživatel těchto formulářů nemůže mít v jednu chvíli otevřených více než jeden[5]. Předpokladem pro toto chování byla úvaha, že uživatel pravděpodobně nebude chtít porovnávat výsledky dvou porovnání.

Jako referenční vnitřní reprezentace, ke které bude probíhání probíhat, je vždy aktuálně zobrazený dokument. Druhou strukturu, která bude porovnávána s tou referenční, je již otevřený dokument, který musí uživatel vybrat prostřednictvím dialogového okna.

Pro potřeby tohoto programu jsem navrhl tři situace, které mohou nastat v závislosti na tom, co uživatel zrovna potřebuje porovnat:

- Porovnání kompletních struktur dvou dokumentů - zobrazí se formulář, ve kterém budou vybrány kořenové uzly obou reprezentací a porovnání proběhne automaticky.
- Porovnání dvou konkrétních uzlů vybraných dokumentů - zobrazí se formulář, ve kterém uživatel nejprve vybere dva uzly k porovnání a proces porovnání spustí ručně. Pokud uživatel vybere k porovnání dva uzly typu *FileSetBlock*, pak je výsledek stejný jako v případě porovnání celé struktury dokumentů.
- Porovnání dvou konkrétních uzlů v rámci jednoho dokumentu - stejné jako v předchozím případě, jen s tím rozdílem, že referenční a porovnávaná struktura jsou shodné.

[5]Toto omezení by se v případě nutnosti dalo obejít buď to dočasně spuštěním nové instance zobrazovacího nástroje nebo trvale změnou dědičnosti formuláře z *JDialog* např. na *JFrame*.

Dva uzly vnitřní reprezentace nemohou být považovány za stejné, pokud je splněna alespoň jedna z následujících podmínek:

- Liší se jejich typ. V tuto chvíli nemá smysl pokračovat v jejich dalším porovnávání.
- Hodnoty stejně pojmenovaných atributů nejsou shodné.
- Shodně pojmenované reference se odkazují každá na uzel jiného typu.
- Liší se počet definovaných objektů v rámci jedné definice stejného jména.

Porovnávání nebere v potaz identifikátory objektů, protože ty se po každém překladu zdrojového souboru liší.

4.5.4 Použité třídy a komponenty

Jazyk Java nabízí nepřeberné množství prvků, ze kterých je možno vystavět funkčně naprosto shodné programy. Nebudu zde uvádět všechny třídy, které jsem během tvorby zobrazovacího nástroje použil, ale jen ty nejdůležitější:

- *HashSet* a *TreeSet* - jedná se o jeden z typů kolekce, konkrétně o implementace množiny. Každý prvek do ní může být vložen jen jednou, pokud by došlo k pokusu o vložení již přítomného prvku, byl by původní prvek nahrazen nově vloženým. Pro programátora je hlavní rozdíl v tom, že prvky množiny *TreeSet* jsou řazeny abecedně.
- *HashMap* a *TreeMap* - podobně jako v předchozím případě jde o kolekce, tentokrát typu mapa. Prvky jsou zde ukládány ve formě dvojic, kdy první člen dvojice je klíč a druhý člen je hodnota klíči odpovídající. Opět nelze uchovávat duplicity klíčů - stejný prvek pro různé hodnoty klíčů však vložen být může. Podrobnější popis různých typů kolekcí včetně ukázkových příkladu je uveden v [5].
- *Vector* - tato třída se podobá množině, ale narozdíl od ní se zde jeden prvek může nacházet i několikrát.
- *JFrame* - jedná se o základ pro tvorbu formulářů, umožňuje jejich pohodlnou tvorbu.
- *JDialog* - chová se podobně jako *JFrame*, ale narozdíl od něj poskytuje možnost tzv. modality formulářů (více dále v části 4.6.6).
- *JPanel* - grafická komponenta umisťovaná na formulář. Chová se jako kontejner, což znamená, že může obsahovat další grafické prvky. Tuto třídu jsem použil jako základ pro grafickou reprezentaci uzlů IR.
- *JInternalFrame* - objekty této třídy se chovají velmi podobně jako *JFrame*, jsou však umisťovány do komponenty *JDesktopPane*, která je poté přidána do *JFrame*.

4.6 Implementační detaily

V této části bude čtenář obeznámen se strukturou programu, a to z pohledu vytvořených tříd a jejich implementace. Tento návrh jsem se pokusil z velké části formulovat předem, ovšem během implementace bylo nutno jej postupně upravovat a doplňovat o další, dříve nedefinované třídy. Všechny třídy jsem, z důvodu jejich nepříliš vysokého počtu, umístil do jediného balíku pojmenovaného *suifViewer*.

V této kapitole nebudu uvádět kompletní popis implementace zobrazovacího nástroje, ale spíše stručný přehled jednotlivých tříd včetně jejich nejdůležitějších částí z hlediska funkcionality. Při vytváření GUI mi byla významným pomocníkem publikace [4], na kterou si čtenáře dovolím odkázat, protože popis vytváření formulářů či zpracování událostí je mimo rámec této práce.

4.6.1 Třída *Main*

Jak už název napovídá, jedná se o třídu, jejíž instance bude vytvořena jako první v celém programu, protože jako jediná obsahuje metodu *main*. Po svém spuštění vytvoří instanci třídy *FrameMain*, které posléze předá řízení programu.

4.6.2 Společné funkce

Ve třídě *Common* jsou definovány metody umožňující zobrazování jednoduchých dialogových oken. Jedná se o metody:

- *showInfo*
- *showWarning*
- *showError*

Třída dále obsahuje jedinou veřejnou konstantu *BACKGROUND_COLOR*, která určuje barvu výplně pozadí formulářů typu *FrameMain* a *FrameDocument*.

4.6.3 Hlavní formulář programu

Instance této třídy je vytvořena ihned při spuštění programu a statická reference na ni zůstává uložena ve třídě *Main*, kde k ní mají přístup všechny ostatní třídy.

FrameMain rozšiřuje třídu *JFrame*. Jak již bylo definováno v části 4.4.1, musím zde uchovávat seznam otevřených dokumentů. K tomuto účelu jsem vytvořil pomocnou třídu *DocumentList*, která se o toto stará. V podstatě se jedná o třídu, která obaluje instanci třídy typu *TreeMap*. V tomto objektu je klíčem název souboru a odpovídající hodnotou reference na instanci třídy *FrameDocument*, popsanou dále v části 4.6.4.

Hlavní formulář programu také obsahuje menu, kde jednotlivé položky umožňují otevírání souborů a práci s již otevřenými dokumenty. Reference na dokument, který je právě aktivní, si udržuji v proměnné *currentDocument*.

Dále zde uchovávám odkaz na formulář pro výběr souboru k porovnání, protože nemá příliš význam jej udržovat pro každý dokument nebo jej dokonce pokaždé vytvářet znovu. Proměnná, která tento odkaz udržuje, má název *frmCompareSelect* a tento formulář je zobrazen vždy, když uživatel potřebuje porovnat dva otevřené soubory. Ostatní akce, které se týkají pouze aktuálního dokumentu, jsou zajištěny voláním jeho odpovídajících veřejných metod.

4.6.4 Tvorba formuláře dokumentu

Jedná se o stěžejní část celého programu, protože v tomto formuláři dochází k vlastnímu vykreslování kompletně načtené struktury vnitřní reprezentace, což bylo vlastně hlavním cílem této práce.

Pokud uživatel vyvolá událost otevření souboru, je vytvořena nová instance třídy *FrameDocument*, která v rámci svého konstruktoru vytvoří GUI a pokusí se nejprve načíst zjednodušenou strukturu zadaného XML souboru. K tomuto účelu jsem si zavedl třídu *SimplifiedXML*, která načte všechna data ze souboru do paměti, kde proběhne jejich transformace. Pro uchovávání XML značek jsem zavedl ještě třídu *SimplifiedXMLNode*, kde jeden objekt tohoto typu reprezentuje jednu značku XML dokumentu. Tato transformace dodržuje následující pravidla:

- Ignoruje veškeré XML komentáře.
- Ignoruje značky *entry*. Tyto značky představují hodnotu v tabulce symbolů a nesou pouze informaci o identifikátoru objektu a jeho názvu. Jelikož se tyto informace dají pohodlněji přečíst i v jiných částech struktury a také z toho důvodu, že si uživatel může sám nadefinovat, ze kterého místa se mají názvy konkrétních objektů načítat (viz. kapitola 3), nemá význam tyto informace dále zpracovávat.
- Pokud má XML značka definovánu textovou položku, bude tato uložena do atributu *text*.
- Každá XML značka má definován nejvýše jeden atribut, přičemž je pro další použití podstatná pouze jeho hodnota, nikoliv název. Tato hodnota je uložena v atributu *attribute*.
- Název XML značky je uložen do proměnné *name*.

Pokud byl celý obsah XML souboru úspěšně zpracován, je nyní v paměti načtena celá struktura vnitřní reprezentace ve zjednodušeném XML formátu. Dalším krokem je podle popisu uvedeného v části 4.5.1 vytvoření objektů představujících vnitřní reprezentaci. Každý jeden objekt, který je definován v této struktuře, odpovídá jedné instanci třídy *SuifNode*.

Nejprve je ze zjednodušené struktury XML souboru načten kořenový prvek, který bude zároveň tvořit také kořenový objekt stromu definic v načítané vnitřní reprezentaci. Třída *Processing* obsahuje jedinou veřejnou metodu, a to *processStructure*. Této metodě je poté předán kořenový prvek dokumentu, odkaz na tabulku uzlů a typů, které jsou

obě uchovávány ve formuláři dokumentu jako *nodes* a *types*. Tato metoda poté zavolá metodu *processNode*, která zahájí rekurzivní zpracování dat z XML dokumentu, přičemž se řídí následujícími pravidly:

- Přeskakuje prvky typu *ViewerAnnote*, protože tyto se do výsledného zobrazení nepromítnou. Jsou z nich pouze přímo načítány informace vztahující se k definici uzlu vnitřní reprezentace.
- Pokud název prvku obsahuje tečku, pak se může jednat o:
  - **Atribut objektu** - může být jednoduchý (např. textová hodnota), jeho hodnota je pak uložena jako atribut. Také se může vyskytnout výčet hodnot, který je ve zjednodušené struktuře uložen jako posloupnost prvků *elem* vnořených do aktuálního prvku (např. pro definici statické metody může jít o prvky *public* a *static*). Každý prvek typu *elem* má svou hodnotu uloženou jako text.
  - **Definici nového objektu** - v tomto případě je do aktuálního prvku zanořena kompletní definice uzlu včetně všech potřebných informací. V tuto chvíli dochází k novému volání metody *processNode* pro každou nalezenou definici.
  - **Referenci na objekt** - v tomto okamžiku ještě nemusím být schopen rozpoznat všechny objekty, proto si případné reference ukládám jako atribut. Reference jsou v XML struktuře zachyceny naprosto stejně jako definice atributu.
- Jestliže byly vyloučeny všechny výše uvedené případy, pak je aktuální prvek považován za definici uzlu vnitřní reprezentace. Pokud již existuje odpovídající prvek typu *SuifNodeType*, pak jej pouze připojím k nově vytvořenému objektu. Pokud se jedná o první výskyt objektu tohoto typu, je vytvořen a uložen do tabulky *types*. Uzel vnitřní reprezentace je poté uložen do tabulky *nodes*.

Jakmile je struktura vnitřní reprezentace načtena podle výše uvedených pravidel, je poslední nutnou operací doplnění referencí a načtení jmen objektů z uživatelem definovaných uzlů. To se provádí ještě před ukončením metody *processStructure* tak, že znova projdu všechny načtené uzly vnitřní reprezentace a pro každý uzel zavolám jeho metodu *processReferences*. Tato metoda projde všechny atributy daného uzlu a pokud zjistí, že některý atribut odpovídá jinému uzlu, jsou tyto reference opraveny.

Dále je nutné doplnit informace o dědičnosti do všech typů uzlů vnitřní reprezentace tak, že odkazy všechny uzly, které daný typ definuje, předám i jeho hierarchicky přímo nadřízenému typu. To v praxi znamená, že např. typ *VariableSymbol* bude obsahovat také odkazy na uzly typu *ParameterSymbol* či *StaticFieldSymbol*.

Po doplnění všech důležitých informací je tedy ve formuláři dokumentu, v tabulkách *nodes* a *types* načtena kompletní struktura dané vnitřní reprezentace, včetně všech definic, referencí a atributů. Navíc je do proměnné *rootSuifNode* uložen odkaz na kořenový uzel celého dokumentu, který je vždy typu *FileSetBlock*. Nyní již tedy mám všechny informace, které potřebuji, abych mohl přistoupit k samotnému zobrazení struktury.

4.6.5 Grafické zobrazení uzlů vnitřní reprezentace

Způsob vykreslení jednotlivých uzlů vnitřní reprezentace byl definován v části 4.5.2, zde jen doplním, že aktuálně vybraný uzel bude vykreslen s červeným textem a okrajem při bílém pozadí. Uzly nevybrané budou mít barvu okraje a textu černou na šedém pozadí.

Pro grafickou reprezentaci jednoho uzlu IR může být obecně vytvořeno několik různých objektů, protože každý uzel vnitřní reprezentace může vystupovat v různých rolích, může být několika uzly referován apod. Z důvodu zjednodušení práce s těmito grafickými objekty bude každé použití uzlu znázorněno jedním unikátním objektem, a sice instancí třídy *GSuifNode* dědící z třídy *JPanel*.

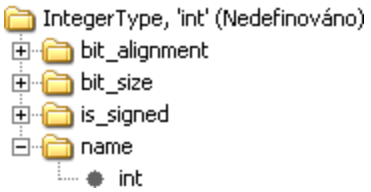
Každý objekt typu *GSuifNode* bude obsahovat odkaz na odpovídající *SuifNode*, který drží informace společné pro všechny jeho grafické podoby. Při vytvoření grafického uzlu je na něj ve výchozím nastavení nahlíženo jako na jeho skutečný typ. Toto nahlížení je možno pro každý zobrazený uzel vnitřní reprezentace změnit na kterýkoli z jemu hierarchicky výše postavených typů. Pro lepší představu jsem přiložil obrázek 9, který zobrazuje dvě různá nahlížení na uzel typu *ClassType*.

Obrázek 9: Příklad různého nahlížení na tentýž uzel IR

Vykreslování probíhá v metodě *paint* třídy *FrameDocument*, ve které je nutné pokaždé znova vykreslit všechny spojnice mezi zobrazenými uzly vnitřní reprezentace. Pro uchování souřadnic každé spojnice jsem musel vytvořit třídu *GLine*, jejíž instance si poté odpovídající *FrameDocument* drží v objektu s názvem *lines*.

Každá spojnice je vykreslena mezi uzlem a jeho zobrazeným poduzlem, ať už definovaným či referovaným, pak pro všechny poduzly a jejich zobrazené poduzly atd. Všechny uzly vnitřní reprezentace, které uživatel vybere k zobrazení, jsou vykresleny pod sebou jako samostatné stromy definic.

Každý zobrazený uzel vnitřní reprezentace lze vybrat klepnutím myši, přičemž jsou všechny jeho vlastnosti, s ohledem na aktuální způsob nahlížení, zobrazeny v komponentě typu *JTree* v pravé části formuláře *FrameDocument*. Kořenovým uzlem takto zobrazených vlastností je pak název aktuálně vybraného uzlu, jeho přímými potomky jsou názvy vlastností, ke kterým jsou přímo připojeny jejich hodnoty, pokud ovšem mají nějaké definovány. Příklad zobrazení vlastností vybraného uzlu je uveden na obrázku 10.



Obrázek 10: Příklad vlastností vybraného uzlu IR

4.6.6 Dialogová okna programu

Abych uživateli usnadnil práci s načtenou vnitřní reprezentací, vytvořil jsem dialogová okna, která si od uživatele před dalším pokračováním požadované akce vyžádají doplňkový vstup. Jedná se o formuláře:

- **Formulář výběru uzlů pro zobrazení** - pokud nepočítám dialog pro otevření souboru, pak se jedná o první dialogové okno celého programu, které uživatel uvidí. Je rozděleno na dva oddělené seznamy, kde vlevo je seznam dostupných typů, které byly pomocí anotace označeny jako zobrazitelné (viz. kapitola 3). V pravém seznamu se pak nacházejí dostupné uzly, které jsou filtrovány podle výběru typů. Po potvrzení dialogu jsou všechny označené uzly zobrazeny ve svém výchozím nastavení.
- **Formulář výběru zobrazených poduzlů** - v tomto dialogovém okně má uživatel možnost dodatečně nastavit zobrazení poduzlů aktuálně vybraného uzlu vnitřní reprezentace. Uzly, které uživatel v tomto seznamu vybere, budou zobrazeny, ostatní budou skryty.
- **Formulář změny způsobu nahlížení** - toto dialogové okno umožňuje uživateli změnit způsob, jakým je nahlíženo na aktuálně vybraný uzel (viz. podrobněji v části 4.6.5).
- **Formulář výběru souboru k porovnání** - poslední dialogové okno celého programu je vyvoláno ve chvíli, kdy uživatel potřebuje porovnávat dvě různé vnitřní reprezentace. Jedinou podmínku je, že porovnávaný dokument již musí být v programu otevřen. Detaily procesu porovnávání jsou uveden v části 4.5.3.

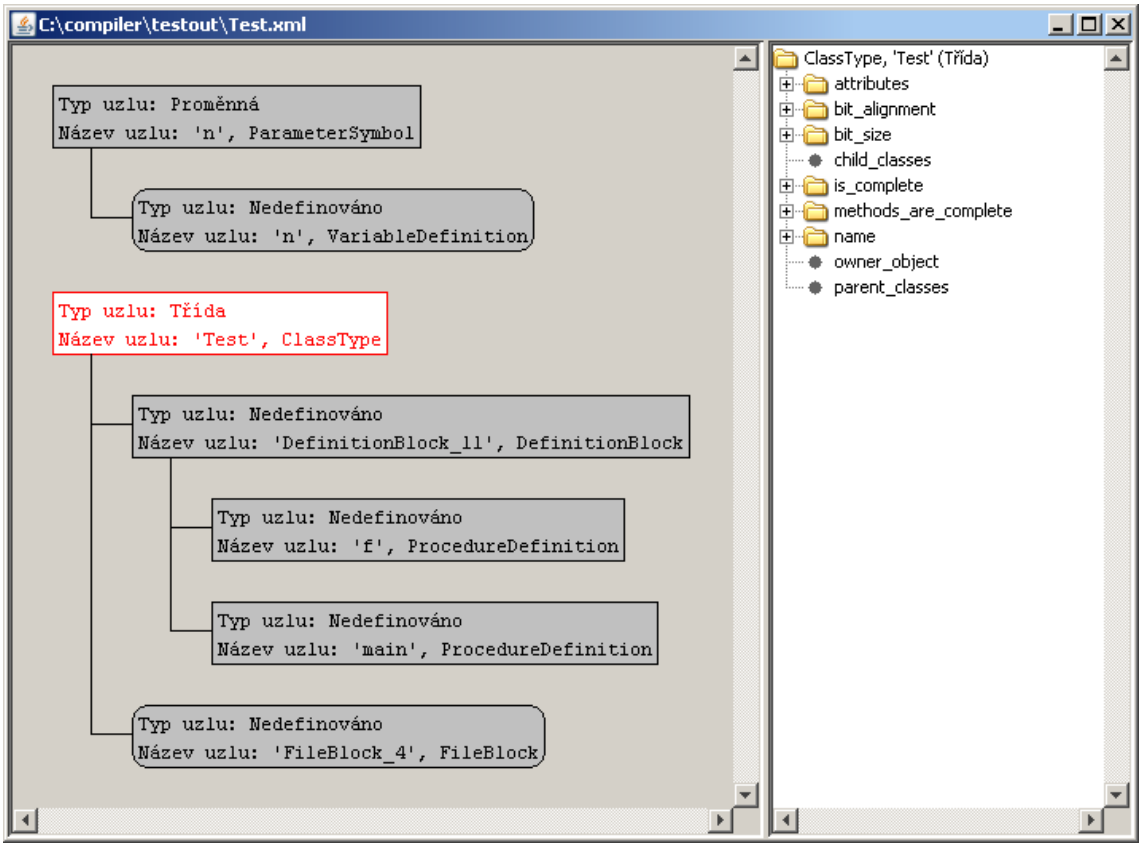
Všechna zmíněná dialogová okna dědí ze třídy *JDialog* a jsou zobrazena modálně. To znamená, že uživatel nemůže s programem dále pracovat, dokud nepotvrdí nebo nestornuje právě zobrazený dialog.

4.7 Příklad zobrazení vnitřní reprezentace

V části 3.3 byl uveden příklad definice tříd v jazyce Hoof při současném použití anotací *@SuifNode*. Ve zmíněné části je také popsán jejich vliv na zobrazované informace. Pro lepší představu výsledného zobrazení jsem navíc ke třídě *ClassType* z modulu *osuif* připojil anotaci uvedenou ve výpise 9. Tato anotace způsobí, že uzly zmíněného typu budou považovány za třídu a ve výchozím nastavení budou zobrazeny všechny objekty definované v *procedure_definitions* uvnitř bloku *definition_block*, včetně tohoto bloku samotného.

```
@SuifNode(display=true, type=SuifNodeConst.TYPE_CLASS, defaultNodes={"definition_block.
    procedure_definitions"})
```

Výpis 9: Anotace třídy *ClassType*



Obrázek 11: Ukázka zobrazení vnitřní reprezentace

Na obrázku 11 je zachyceno možné zobrazení uzlů vnitřní reprezentace. Aktuálně vybraný uzel je typu *ClassType* a má zobrazeny dva poduzly - referovaný typu *FileBlock* a definovaný typu *DefinitionBlock*, který má navíc zobrazeny ještě další dva poduzly v něm definované (oba typu *ProcedureDefinition*).

Nad definicí třídy je ještě zobrazen uzel vnitřní reprezentace typu *ParameterSymbol*, která má zobrazen referovaný poduzel typu *VariableDefinition*.

Z obrázku je také patrné, že uživatel nepoužil anotace *@SuifNode* k nastavení jmen uzlů typu *DefinitionBlock* a *FileBlock*, které mají v této chvíli název vygenerován zobrazovacím nástrojem.

5 Závěr

Nástroj vytvořený v rámci této bakalářské práce se snaží z vygenerované vnitřní reprezentace vypreparovat všechny důležité informace a tyto informace co možná nejjednodušším a přitom přehledným způsobem zobrazit, což se mi, jak pevně věřím, podařilo.

Jedním z mých cílů bylo také seznámit čtenáře se základy systému SUIF, stručně popsat jeho význam a možnosti a dát čtenáři k dispozici dostatek informací, aby byl schopen s tímto systémem pracovat a upravovat již stávající sadu tříd podle svých potřeb.

Nesnažím se tvrdit, že způsob řešení, který jsem zde prezentoval, je jediný správný, jsem si jist, že existují i jiné a elegantnější cesty, jak dospět ke stejnému či podobnému výsledku.

Jsem přesvědčen, že zadání této bakalářské práce bylo úspěšně splněno a případné drobné nedostatky nebudou příliš bránit jejímu kladnému přijetí.

Svatoslav Musil

6 Reference

[1] Aigner, G.; Diwan, A.; Heine, D. L.; aj.: *An Overview of the SUIF2 Compiler Infrastructure*. Computer Systems Laboratory, Stanford University, Portland Group, Inc., 1999. URL `http://suif.stanford.edu/`

[2] Aigner, G.; Diwan, A.; Heine, D. L.; aj.: *The SUIF Program Representation*. Computer Systems Laboratory, Stanford University, The Portland Group, Inc., 2000. URL `http://suif.stanford.edu/`

[3] Běhálek, M.: *Přední část překladače jazyka e-Java*. Diplomová práce, VŠB-TUO Ostrava, 2002.

[4] Herout, P.: *Java - grafické uživatelské prostředí a čeština*. KOPP, 2004.

[5] Herout, P.: *Java - bohatství knihoven*. KOPP, 2006.

[6] Moore, D. L.: *The SUIF Programmer Guide*. The Portland Group, Inc., 1999. URL `http://suif.stanford.edu/`

A Přiložené CD

Tato příloha by měla čtenáři umožnit lepší orientaci ve struktuře přiloženého CD, které obsahuje následující adresáře:

- *compiler2* - obsahuje překladač systému SUIF. Jedná se o zdrojové kódy vytvořené původně pro [3], které jsem pro účely této práce doplnil.
- *javacc* - zde je v komprimované podobě umístěn nástroj *JavaCC 4.1*, který je pro správnou funkci překladače nutno rozbalit na pevný disk počítače a nastavit systémou proměnnou *PATH* tak, aby ukazovala na adresář *bin*. Nejnovější verze je dostupná na *https://javacc.dev.java.net/*
- *jdk* - obsahuje instalaci *Java SE Development Kit 6 (Update 20)*, na kterém byl vyvíjen a testován nástroj pro zobrazování vnitřní reprezentace. Nejnovější verze je dostupná na *http://java.sun.com/*
- *suifViewer* - v tomto adresáři je uložen nástroj pro zobrazování vnitřní reprezentace. Vnořené adresáře jsou:
  - *bin* - obsahuje spustitelný soubor *suifViewer.jar*
  - *src* - obsahuje kompletní zdrojové kódy zobrazovacího nástroje
- *thesis* - text této bakalářské práce ve formátu PDF.

Použití přiloženého CD

Následující postup předpokládá správně nainstalované *Java SDK* a *JavaCC* minimálně ve verzích dostupných z přiloženého CD.

Nejprve je nutno na pevný disk nakopírovat celý adresář *compiler2*. Jeho umístění ani název není podstatný, jen by měla být zachována jeho struktura. Přímo v tomto adresáři jsou umístěny tři dávkové soubory:

- *make.bat* - vygeneruje a přeloží všechny třídy potřebné pro správnou funkci překladače.
- *make_test.bat* - jako předchozí soubor, ale navíc ještě přeloží testovací soubory z adresáře *testfile* a výsledek uloží do adresáře *testout*.
- *make_clean.bat* - odstraní všechny soubory vygenerované pomocí výše uvedených dávkových souborů.

Soubor *suifViewer.jar* je spustitelný Java archiv, ve kterém je uložen celý nástroj na zobrazování vnitřní reprezentace. Přestože je možno jej spouštět přímo z CD, doporučuji jej také zkopírovat na pevný disk.